PSCV: A Runtime Verification Tool for Probabilistic SystemC Models

Van Chan Ngo, Axel Legay, Vania Joloboff

PSCV: A Runtime Verification Tool for Probabilistic SystemC Models

Van Chan Ngo[1], Axel Legay[2], and Vania Joloboff[2]

[1] Carnegie Mellon University, Pittsburgh PA 15213, USA
[2] Inria Rennes - Bretagne Atlantique, Rennes 35042, France

Abstract. This paper describes PSCV, a runtime verification tool for a class of SystemC models which have inherent probabilistic characteristics. The properties of interest are expressed using bounded linear temporal logic. The various features of the tool including automatic monitor generation for producing execution traces of the model-under-verification, mechanism for automatically instrumenting the model, and the interaction with statistical model checker are presented.

1 Introduction

SystemC[3], a C^{++} library [6], has been become increasingly prominent in modeling hardware and embedded systems at the level of transactions. Models can be used to simulate the system behavior with a single-core reference event-driven simulation kernel [2]. A SystemC model is a complex and multi-threaded program where scheduling is cooperative and thread execution is mutually exlusive. In many cases, models include probabilistic characteristics, i.e., random data, reliability of the system's components. Hence, it is crucial to evaluate quantitative and qualitative analyses of system property probabilities. Many algorithms [4,7,10] with the corresponding mature tools based on model checking techniques, i.e., *Probabilistic Model Checking* (PMC), are created, in which they compute probability by a numerical approach. However, they are infeasible for large real-life systems due to *state space explosion* and cannot work directly with SystemC source code.

In this paper we present PSCV a new tool for checking properties expressed in *Bounded Linear Temporal Logic* (BLTL) [14] of probabilistic SystemC models. It uses *Statistical Model Checking* (SMC) [7,17,8,16,14,9,18] techniques, a simulation-based approach. Simulation-based approaches use a finite set of system executions to produce an approximation of the value to be evaluated. Since these techniques do not construct all reachable states of the model-under-verification (MUV), execution time and memory space required are far less than numerical approaches.

The tool supports a rich set of properties, a wide range of abstractions from statement level to system level, and a more fine-grained model of time than

[3] IEEE Standard 1666-2005

a coarse-grained cycle-based simulation provided by the current SystemC kernel [2]. Given a property, a user-defined absolute error and confidence, the tool implements the *statistical estimation* and *hypothesis testing* techniques [8,16] for computing probability that the property is satisfied by the model or asserting that this probability is at least equal to a threshold. The theoretical and algorithmic foundations of the tool are based on Ngo et al.'s work [12].

2 Verification Flow

The verification flow using PSCV consists of three steps, as shown in Fig. 1, in which the *Monitor and Advice Generator* (MAG), AspectC^{++}, the modified SystemC kernel, and SystemC plugin are components of PSCV. In the first step, users write a configuration file containing a set of typed variables called *observed variables*, a Boolean expression called *temporal resolution*, and all properties to be verified. MAG translates the configuration file into a C^{++} monitor and a set of aspect-advices. In the second step, the set of aspect-advices is used as an input of AspectC^{++} to automatically instrument the MUV for exposing the user model states and syntax. The instrumented model and the generated monitor are compiled and linked together with the modified SystemC kernel to produce an executable model.

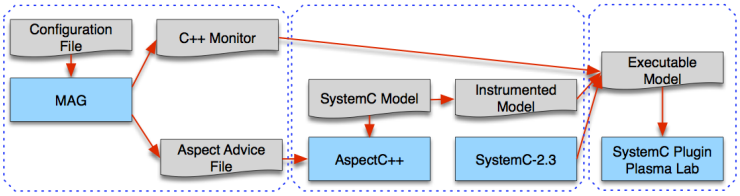

Fig. 1: The verification flow

Finally, the SystemC plugin independently simulates the executable model in order to make the monitor produce execution traces with inputs provided by the user. The inputs can be generated using any standard stimuli-generation technique. These traces are finite in length since the BLTL semantics [14] is defined with respect to finite execution traces. The number of simulations is determined by the statistic algorithm used by the plugin. Given these execution traces and the user-defined absolute error and confidence, the SystemC plugin employs SMC to produce an estimation of the probability that the property is satisfied or an assertion that this probability is at least equal to a threshold.

3 Expressing Properties

The tool accepts input properties of the forms $\Pr(\varphi)$, $\Pr_{\geq\theta}(\varphi)$, and $X_{\leq \mathrm{T}}(rv)$, where φ is a BLTL formula. The first is used to compute the probability that

φ satisfied by the model. The second asserts that this probability is at least equal to the threshold θ. The last returns the mean value of random variable rv. The set of atomic propositions in the logic describes SystemC code features and the simulation semantics. It is a set of Boolean expressions defined over a set of typed variables called *observed variables* with the standard operators $(+, -, *, /, >, \geq, <, \leq, !=, =)$. The semantics of the temporal operators in BLTL formulas interpreted over states is defined by a *temporal resolution* that defines at which time points the states are sampled in order to make the transition from one state to another state. A temporal resolution is a logical disjunction over a set of Boolean observed variables, in which the tool should sample a new state whenever the temporal resolution is evaluated to true. For example, assume that we want the satisfaction of the underlying formula φ to be checked either at the end of every delta-cycle or every time immediately after the event e is notified. Hence, the temporal resolution is defined by the following disjunction (MON_DELTA_CYCLE_END | e.notified), where MON_DELTA_CYCLE_END and e.notified are Boolean observed variables that have the value *true* whenever the kernel phase is at the end of delta-cycle and e is notified, respectively. The observed variables used to describe SystemC code features, the simulation semantics, and temporal resolution are summarized below; see [13,12] for the full syntax and semantics.

Attribute. Users can define an observed variable whose value and type are equal to the value and type of a module's attribute in the user code. Attributes can be public, protected, or private. For example, $a.t$ a_t defines a variable named a_t whose value and type are equal to the value and type of the private attribute t of the module instance a.

Function. Let f be a C^{++} function with k arguments in the user code. Users can refer to locations in the source code that contain the function call, immediately after the function call, immediately before the first executable statement, and immediately after the last executable statement in f by using the Boolean observed variables $f()$:call, $f()$:return, $f()$:entry, and $f()$:exit, respectively. Moreover, users define an observed variable $f()$:i, $i = 0, \cdots, k$, whose value and type are equal to the value and type of the return object (with $i = 0$) or i^{th} argument of function f before executing the first statement in the function body. For example, if the function int div(int x, int y) is defined in the user code, then the formula $G_{\leq T}(div()$:entry $\rightarrow div()$:2 != $0)$ asserts that the divisor is nonzero whenever the div function starts execution.

Simulation phase. There are 18 predefined Boolean observed variables which refer to the 18 kernel states [13]. These variables are usually used to define a temporal resolution. For example, the formula $G_{\leq T}(p = 0)$ which is accompanied by the temporal resolution (MON_DELTA_CYCLE_END) requires the value of variable p to be zero at the end of every delta-cycle.

Event. For each SystemC event e, PSCV provides a Boolean observed variable e.notified that is true only when the simulation kernel actually notifies e. For example, the formula $G_{\leq T}(e.\text{notified} \rightarrow (a = 0))$ says that whenever the event e is notified, a equals to 0.

4 Architecture

PSCV, available as an open-source software [13], implements SMC for probabilistic SystemC models. The main components are depicted in Fig. 2. It consists of off-the-self, modified and original components: (1) an off-the-self component, AspectC^{++} [5], a C^{++} aspect compiler for instrumenting the MUV, (2) a modified component, a patched SystemC-2.3.0 kernel for facilitating the communication between the kernel and the monitor and implementing a random scheduler, and (3) two original components are MAG, a C^{++} tool for automatically generating monitor and aspect-advices for instrumentation, and SystemC plugin, a plugin of the statistical model checker Plasma Lab [3].

4.1 Execution Trace Extraction

In PSCV, based on the techniques in [15], the set of observed variables and temporal resolution are converted into a C^{++} monitor class and a set of aspect-advices. MAG generates three files: aspect_definitions.ah, monitor.h, and monitor.cc, in which they contain a set of AspectC^{++} *aspect* definitions, one monitor class, and a class called *local_observer* that is responsible for invoking the callback functions, which invoke the sampling function at the right time point during the MUV simulation.

The monitor has a *step* function, sampling function, that waits for a request from the SystemC plugin. If the request is stopping the current simulation, it then terminates the MUV execution. If the plugin requests a new state, then the current values of all observed variables and the simulation time are sent. The *step* function is called at every time point defined by temporal resolution. These time points can be kernel phases, event notifications, or locations in the MUV code control flow. In such cases, the patched kernel needs to communicate with the *local_observer*, i.e., when a delta-cycle ends, via a class called *mon_observer* to invoke the *step* function of the monitor. In case of locations in the MUV code, the advice code generated by MAG will call the callback function to invoke the *step* function.

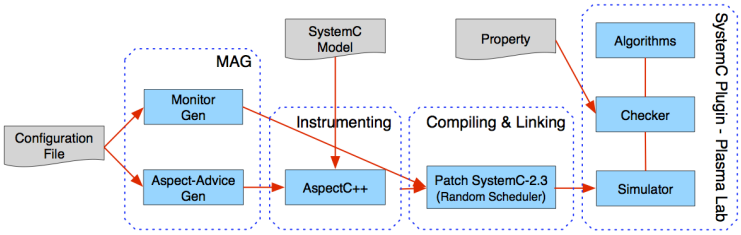

Fig. 2: The architecture of PSCV

The aspect is an extension of the class concept of C^{++}, to collect advice code implementing a common crosscutting concern in a modular way. For example,

to access all attributes of a module called A and the location that occurs immediately before the first executable statement of the function foo in A (defined in the configuration file as % A::foo():entry), MAG generates the following aspect definition.

```
aspect Automatic {
  pointcut reveal() = "A"; //Pointcut for accessing private data of A
  advice reveal() : slice class {
    friend class monitor_A; //Generated monitor is friend class of A
  };
  advice execution("% A::foo()"): before() { //Instrumentation code
    mon_observer* observer = local_observer::getInstance();
    monitor_A* mon = (monitor_A*) observer->get_monitor_by_index(0);
    mon->callback_userloc_loc1(); //Invoke callback function
  }
};
```

4.2 Statistical Model Checker

The statistical model checker is implemented as a plugin of Plasma Lab. It establishes a communication, in which the generated monitor transmits execution traces of the MUV. In the current version, the communication is done via the standard input and output. When a new state is requested, the monitor reports the current state containing current observed variable values and the simulation time to the plugin. The length of traces depends on the satisfaction of the formula, which is finite due to the bounded temporal operators. Similarly, the required number of traces depends on the statistic algorithms in use.

4.3 Random Scheduler

Verification does not only depend on the probabilistic characteristics of the MUV, but it also can be significantly affected by the *scheduling policy*. Consider a simple module A that consists of two thread processes as shown in the following listing, where x is initialized to be 1.

```
void A::t1() {
  if (x <= 0)
    x := x + 1;
}

void A::t2() {
  if (x > 0)
    x := x - 1;
}

SC_CTOR(A) {
  SC_THREAD(t1);
  SC_THREAD(t2);
}
```

Assume that we want to compute the probability that x is always equal to 1. Obviously, x depends on the execution order of two threads, i.e., the value is 1 if $t2$ is executed before the execution of $t1$ and 0 if the order is $t1$ then $t2$. The current scheduling policy is deterministic as it always picks the process that is first added into the queue, the implementation uses a queue to store a set of runnable processes. Hence, only one execution order, $t1$ then $t2$, is verified instead of two possible orders. As a result, the probability to be verified is 0, however, it should be 0.5. Therefore, it is more interesting if a verification is performed on all possible execution orders than a fixed one. In many cases, there is no decision or an a-priori knowledge of the scheduling to be implemented. Moreover, verification

of a specification should be independent of the scheduling policy to be finally implemented.

To perform our verification on possible execution orders of the MUV, we implemented a random scheduler. The source of the process scheduler is the evaluation phase, in which one of the runnable processes is executed. Given a set of N runnable processes in the queue at the evaluation phase, the scheduler randomly chooses one of these processes to execute. The random algorithm is implemented by generating a random integer number uniformly over a range $[0, N-1]$. For more simulation efficiency and implementation simplicity, we employ the *rand*() function and % operator in C/C^{++}.

5 Experimental Results

We report the experimental results for several examples including a running example, a case study of dependability analysis of a large control system (i.e., the number of states is 2^{155}), and random scheduler examples. We used the 2-sided Chernoff bound with the absolute error $\varepsilon = 0.02$ and the confidence $\alpha = 0.98$. The experiments were run on machine with Intel Core i7 2.67 GHz processor and 4GB RAM under the Linux OS. The analysis of the control system takes almost 2 hours, in which 90 liveness properties were verified. The full experiments can be found at the website [13]. For example, the first property

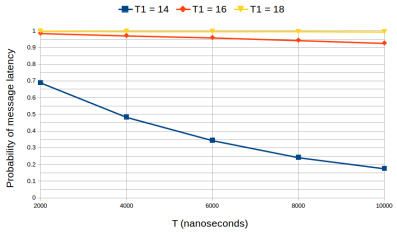

Fig. 3: Message latency

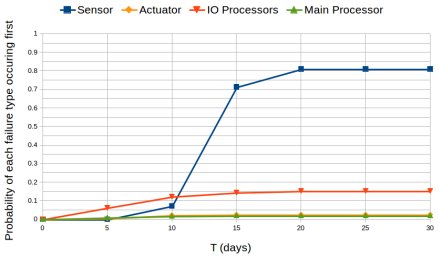

Fig. 4: Each component fails first

we checked is the probability that the message latency from the producer to the consumer within T_1 time units over a period of T time of operation using the formula $\varphi = \mathrm{G}_{\leq \mathrm{T}}((\mathrm{c\_read} = \ '\&') \rightarrow \mathrm{F}_{\leq \mathrm{T}_1}(\mathrm{c\_read} = \ '@'))$, where c_read, &, and @ are the current received character, special starting and ending delimiters, respectively. The second property we tried to determine which kind of component is more likely to cause the failure of the control system. It is expressed in BLTL as $\neg$shutdown $\mathrm{U}_{\leq \mathrm{T}}$ $\mathrm{failure_i}$, where shutdown $= \bigvee_{i=1}^{4} \mathrm{failure_i}$. The results are plotted in Fig. 3 and Fig. 4.

For the random scheduler, it seems that the implementation with the pseudo random number generator (PRNG), by using the *rand*() function and % opera-

tor is not efficient. We are planning to investigate the Mersenne Twister generator [11] that is by far the most widely used general-purpose PRNG in order to deal with this issue.

6 Conclusion

We present PSCV an SMC-based verification tool for checking properties expressed in BLTL of probabilistic SystemC models. The tool supports a rich set of properties, a wide range of abstractions from statement level to system level, and a more fine-grained model of time. In the feature we plan to make the verification process more automated by eliminating the user interaction with AspectC^{++} and embedding the checker inside the tool such as [1].

References

1. Abarbanel, Y., Beer, I., Glushovsky, L., Keidar, S., Wolfsthal, Y.: Focs: Automatic generation of simulation checkers from formal specifications. In: CAV (2000)
2. Accellera: `http://www.accellera.org/downloads/standards/systemc`
3. Boyer, B., Corre, K., Legay, A., Sedwards, S.: Plasma Lab: A flexible, distributable statistical model checking library. In: QEST (2013)
4. Ciesinski, F., Grober, M.: On probabilistic computation tree logic. In: Validation of Stochastic Systems (2004)
5. Gal, A., Schroder-Preikschat, W., Spinczyk, O.: AspectC++: Language proposal and prototype implementation. In: OOPSLA (2001)
6. Grotker, T., Liao, S., Martin, G., Swan, S.: System Design with SystemC. Kluwer Academic Publishers (2002)
7. Hermanns, H., Watcher, B., Zhang, L.: Probabilistic Cegar. In: CAV (2008)
8. Hoeffding, W.: Probability inequalities for sums of bounded random variables. In: American Statistical Association (1963)
9. Katoen, J., Hahn, E., Hermanns, H., Jansen, D., Zapreev, I.: The Ins and Outs of the probabilistic model checker MRMC. In: QEST (2009)
10. Kwiatkowska, M., Norman, G., Parker, D.: Controller dependability analysis by probabilistic model checking. In: Control Engineering Practice (2007)
11. Matsumoto, M., Nishimura, T.: Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. In: ACM Transactions on Modeling and Computer Simulation (1998)
12. Ngo, V.C., Legay, A., Quilbeuf, J.: Statistical model checking for SystemC models. In: HASE (2016)
13. PSCV: `https://project.inria.fr/pscv/` (2016)
14. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: CAV (2004)
15. Tabakov, D., Vardi, M.: Monitoring temporal SystemC properties. In: Formal Methods and Models for Codesign (2010)
16. Younes, H.: Verification and planning for stochastic processes with asynchronous events. In: PhD Thesis, Carnegie Mellon (2005)
17. Younes, H., Kwiatkowska, M., Norman, G., Parker, D.: Numerical vs statistical probabilistic model checking. In: STTT (2006)
18. Zuliani, P., Platzer, A., M.Clarke, E.: Bayesian statistical model checking with application to Simulink/Stateflow verification. In: FMSD (2013)